Chapter FOURTEEN
Optional Class

Exam Objectives

Develop code that uses the Optional class.

The problem with null

Most programming languages have a data type to represent the absence of a value, and it is known by many names:

```
NULL, nil, None, Nothing
```

The `null` type was introduced in ALGOL W by Tony Hoare in 1965, and it's considered one of the worst mistakes of computer science. In Tony Hoare's own words:

I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object-oriented language ((ALGOL W)). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.
– Tony Hoare

Still, some may be wondering, what is the problem with `null`?

Well, if you're a little worried by the problems this code might cause, you know the answer already:

```
String summary =
  book.getChapter(10)
          .getSummary().toUpperCase();
```

The problem with that code is that if any of those methods returns a `null` reference (for example, if the book doesn't have a tenth chapter), a `NullPointerException` (the most common exception in Java) will be thrown at runtime stopping the program.

What can we do to avoid this exception?

Perhaps, the easiest way is to check for `null` . Here's one way to it:

```
String summary = "";
if(book != null) {
    Chapter chapter = book.getChapter(10);
    if(chapter != null) {
        if(chapter.getSummary() != null) {
          summary = chapter.getSummary()
                        .toUpperCase();
        }
    }
}
```

You don't know if any object in this hierarchy can be `null` , so you check every object for this value. Obviously, this is not the best solution; it's not very practical and damage readability.

There may be another issue. Is checking for `null` really desirable? I mean, what if those objects should never be `null` ? By checking for `null` , we will be hiding the error and not be dealing with it.

Of course, this is also a design issue. For example, if a chapter has no summary yet, what would be better to use as a default value? An empty string or `null` ?

To address this problem, Java 8 introduced the class `java.util.Optional<T>` .

The Optional class

The job of this class is to **ENCAPSULATE** an optional value, an object that can be `null` .

Using the previous example, if we know that not all chapters have a summary, instead of modeling the class like this:

```
class Chapter {
    private String summary;
    // Other attributes and methods
}
```

We can use the `Optional` class:

```
class Chapter {
    private Optional<String> summary;
    // Other attributes and methods
}
```

So if there's a value, the `Optional` class just wraps it. Otherwise, an empty value is represented by the method `Optional.empty()` , that returns a singleton instance of `Optional` .

By using this class instead of `null` , first, we explicitly declare that the summary attribute is optional. Then, we can avoid `NullPointerExceptions` while having at our disposal the useful methods of `Optional` that we'll review up next.

First, let's see how to create an instance of this class.

To get an empty `Optional` object, use:

```
Optional<String> summary = Optional.empty();
```

If you are sure that an object is not `null`, you can wrap it in an `Optional` object this way:

```
Optional<String> summary = Optional.of("A summary");
```

A `NullPointerException` will be thrown if the object is `null`. However, you can use:

```
Optional<String> summary = Optional.ofNullable("A summary");
```

That returns an `Optional` instance with the specified value if it is non- `null`. Otherwise, it returns an empty `Optional`.

If you want to know if an `Optional` contains a value, you can do it like this:

```
if( summary.isPresent() ) {
    // Do something
}
```

Or in a more functional style:

```
summary.ifPresent(s -> System.out.println(s));
// Or summary.ifPresent(System.out::println);
```

The `ifPresent()` method takes a `Consumer<T>` as an argument that is executed only if the `Optional` contains a value.

To get the value of an `Optional` use:

```
String s = summary.get();
```

However, this method will throw a `java.util.NoSuchElementException` if the `Optional` doesn't contain a value, so it's better to use the `ifPresent(`) method.

Alternatively, if we want to return something when the `Optional` doesn't contain a value, there are three other methods we can use:

```
String summaryOrDefault = summary.orElse("Default summary");
```

The `orElse()` method returns the argument (that must be of type `T`, in this case a `String`) when the `Optional` is empty. Otherwise, it returns the encapsulated value.

```
String summaryOrDefault =
        summary.orElseGet( () -> "Default summary" );
```

The `orElseGet()` method takes a `Supplier<? extends T>` as an argument that returns a value when the `Optional` is empty. Otherwise, it returns the encapsulated value.

```
String summaryOrException =
        summary.orElseThrow( () -> new Exception() );
```

The `orElseThrow()` method takes a `Supplier<? extends X>`, where `X` is the type of the exception to throw when the `Optional` is empty. Otherwise, it returns the encapsulated value.

Like streams, there are versions of the `Optional` class to work with primitives, `OptionalInt` , `OptionalLong` , and `OptionalDouble` , so you can use `OptionalInt` instead of `Optional<Integer>` :

```
OptionalInt optionalInt = OptionalInt.of(1);
int i = optionalInt.getAsInt();
```

However, the use of these primitive versions are not encouraged, especially because they lack three useful methods of `Optional` : `filter()` , `map()` , and `flatMap()` . And since `Optional` just contains one value, the overhead of boxing/unboxing a primitive is not significant.

The `filter()` method returns the `Optional` if a value is present and matches the given predicated. Otherwise, an empty `Optional` is returned.

```
String summaryStr =
    summary.filter(s -> s.length() > 10).orElse("Short summary");
```

The `map()` method is generally used to transform from one type to another. If the value is present, it applies the provided `Function< ? super T, ? extends U >` to it. For example:

```
int summaryLength = summary.map(s -> s.length()).orElse(0);
```

The `flapMap()` method is similar to `map()` , but it takes an argument of type `Function<? super T, Optional<U>>` and if the value is present, it returns the `Optional` that results from applying the provided function. Otherwise, it returns an empty `Optional` .

In Chapter 17, we'll review in more detail the methods `map()` and `flatMap()` and how they are used with streams.

Key Points

- The `java.util.Optional<T>` class **ENCAPSULATES** an optional value, i.e. an object that can be `null` .
- An empty value is represented by the method `Optional.empty()` .
- You can wrap an object in an `Optional` with the method `of()` , however, a `NullPointerException` will be thrown if the object is `null` .
- The method `ofNullable()` returns an `Optional` instance with the specified value if it is non- `null` . Otherwise, it returns an empty `Optional` .
- To get the value of an `Optional` use the method `get()` , but it will throw a `java.util.NoSuchElementException` if the `Optional` doesn't contain a value, so it's better to use the `ifPresent()` method that takes a `Consumer<T>` as an argument that is executed only if the `Optional` contains a value.
- The `orElse()` method returns the argument when the `Optional` is empty, otherwise, it returns the encapsulated value.
- The `orElseGet()` method takes a `Supplier` that returns a value when the `Optional` is empty. Otherwise, it returns the encapsulated value.
- The `orElseThrow()` method takes a `Supplier` that returns an exception when the `Optional` is empty. Otherwise, it returns the encapsulated value.

Self Test

1. Given:

```
public class Question_14_1 {
    public static void main(String[] args) {
        Optional opt = Optional.of("1");
        String s = opt.orElseGet(
            () -> new RuntimeException()
        );
        System.out.println(s);
    }
}
```

What is the result?
A. `1`
B. Nothing is printed
C. Compilation fails
D. An exception occurs at runtime

2. Which of the following statements is true?
A. The method `Optional.isPresent()` takes a `Consumer<T>` as an argument that is executed only if the `Optional` contains a value.
B. The method `Optional.of()` can create an empty `Optional` .
C. The method `Optional.of()` can throw a `NullPointerException` .
D. The method `Optional.ifPresent()` takes a `Function<T,U>` as an argument.

3. Given:

```
public class Question_14_3 {
    public static void main(String[] args) {
        System.out.println(ToInt("a").get());
    }
    private static Optional<Integer> ToInt(String s) {
        try {
            return Optional.of(Integer.parseInt(s));
        } catch(Exception e) {
            return Optional.empty();
        }
    }
}
```

What is the result?
A. `a`
B. `Optional.empty`
C. Compilation fails
D. An exception occurs at runtime

4. Given:

```
public class Question_14_4 {
    public static void main(String[] args) {
        System.out.println(
            Optional.of(0).orElse(1)
        );
    }
}
```

What is the result?
A. `0`
B. `1`
C. Compilation fails
D. An exception occurs at runtime

Open answers page

Do you have a problem or something to say?

Report an issue with the book

Contact me

13. Iterating and Filtering Collections 15. Data Search